On the security of open source software

Christian Payne

School of Information Technology, Murdoch University, Murdoch 6150, Perth, Western Australia, email: christian@it.murdoch.edu.au

Abstract. *With the rising popularity of so-called 'open source' software there has been increasing interest in both its various benefits and disadvantages. In particular, despite its prominent use in providing many aspects of the Internet's basic infrastructure, many still question the suitability of such software for the commerce-oriented Internet of the future. This paper evaluates the suitability of open source software with respect to one of the key attributes that tomorrow's Internet will require, namely security. It seeks to present a variety of arguments that have been made, both for and against open source security and analyses in relation to empirical evidence of system security from a previous study. The results represent preliminary quantitative evidence concerning the security issues surrounding the use and development of open source software, in particular relative to traditional proprietary software.*

Keywords: open source software, proprietary software, computer security, security vulnerabilities, code review

INTRODUCTION

In today's interconnected world, both businesses and individuals are finding that they are increasingly reliant on computer systems and networks. With this increased dependence often comes a highlighted sense of the importance of maintaining control over this information infrastructure. Without this security, businesses are hamstrung by being unable to trust the information systems they depend on for their survival. Furthermore, in the growth area of Internet commerce, lack of customer confidence in the security of online transactions is still regarded as one of the major limiting factors.

So just what sorts of systems are modern Internet users, be they home or business, depending on? Interestingly, more often that not, the systems on which the Internet is built are not exclusively proprietary, commercially developed code. In fact, many of the programs that make up the infrastructure of the Internet are actually available for free. This is known as 'free software' and consists of programs for which the source code is available and for which users are granted certain rights rarely given where commercial code is concerned. Users are not only able to obtain the source code and use the program for free but they are also encouraged to redistribute it to others (whether gratis or for a fee) and even have permission to modify it. In recent years this concept has become increasingly popular and has been given the new name of 'open source software' (Raymond, 2000), (Norin & Stöckel, 1998). However, while this has been the subject of much publicity in recent years, this software has actually been of critical importance in keeping the Internet running virtually since its birth. The BIND name server provides all-important DNS services to the vast majority of Internet users and has done so for years. The SENDMAIL mail transfer agent (MTA) is involved in the delivery of a significant proportion of the millions of email messages that are sent daily. The Apache web (HTTP) server that, in the face of intense competition from Microsoft, still manages to regularly increase its market share. [At the time of writing the percentage of web servers running Apache was 59.99% which represented a 1.24% increase from the previous month (*Netcraft Web Server Survey*, 2001).] And the Unix systems based around Linux and BSD, who's speed and reliability are legendary, are also proving more and more popular as server platforms (DiBona, Ockman & Stone, 1999).

But how good really is all this software? In particular, how secure is it? Certainly the Internet has grown from infancy depending on this free code but today's online world is no longer the domain of academics and hobbyists. Can this 'homegrown' code be relied upon for the business-centric Internet of the future? Is open source software really secure? How does it perform in this department relative to traditional, proprietary software and what special security issues pertain to it given that the source code is continually available to potential attackers?

In recent times there have been many differing opinions presented on this matter. Some argue what seems counter-intuitive to many, that free software is actually more secure. Others disagree, claiming the availability of source code makes things easier for those seeking to compromise systems. Despite the fact that a lot has been said about this topic, few facts have actually been presented. Certainly the opinions of experts carry much weight but, before they can be accepted, empirical evidence must be provided. This paper seeks to analyse the arguments of both sides and assess them in the light of results from an empirical study that measured the security of three different operating systems.

Definitions

Although it is most often known as 'open source software', the software described in this paper was originally more commonly known as 'free software'. The concept of free software is to grant users certain freedoms. In this sense the word 'free' does not mean 'without cost' but rather free in the sense of having freedom. Indeed, the analogy commonly given amongst the free software community is 'think of "free speech", not 'free beer"'. Although the specific licences under which free software is placed often differ in their details, free software will usually grant its users the rights to access the source code for the purposes of modification, re-distribute the software (either modified or otherwise) and sell the software either on its own or as part of another product, and generally does not restrict the purposes for which the software may be used. A strong factor in the development of free software was, and still remains, ideology. The rights of individuals to have access to programs that they need and the freedom to use and manage the code as they see fit are often strongly held beliefs amongst many free software developers. However, the 'open source' movement has resulted in these ideals being somewhat diluted. The emphasis of open source is on improved, cooperative development methodologies and on improving acceptance of free software amongst business. The term itself is also often misleading. Many mistakenly believe that 'open source' simply means that the source code is available whereas this is certainly not the case – there are many pieces of software where the source code is available but users have no rights to modify or redistribute the code. This software is not 'open source' despite the source being 'open'. Software that falls into this category is not the focus of this paper although there are some conclusions concerning it that may be arrived at. As mentioned, there are many different licences that qualify as free software although each has different terms. Two common licences are the General Public Licence (GPL) and BSD licence. There is no universal standard for determining whether a particular licence qualifies as free software, although the Open Source Definition (OSD) and Debian Free Software Guidelines (DFSG) represent two commonly accepted (and very similar) sets of criteria.

Defining the opposite of open source software is much simpler. This traditional means of licensing software is often described as 'proprietary' or 'binary-only' (in reference to the usual lack of source code) although the informal term 'closed source' has proven an intuitive and effective antonym.

Past work

As stated more, a wide array of opinions have been expressed in the popular on-line press concerning this issue (Levy, 2000), (Garfinkel, 1999), (Gross, 2000), (Viega, 2000), (Chowdhry, 1999). However, there has been little discussion of this in the academic literature although the issue was discussed at the first Composable High Assurance Trusted Systems (CHATS) workshop held on 30 November 1999.

Paper organisation

This paper begins by delineating the common arguments that are regularly given by those who claim that open source provides significantly better security than traditional proprietary software. Then the alternative view point is considered. Next a recent study is described which yielded certain empirical results that can be used to provide evidence concerning the question of the security of this sort of software. Finally arguments from both sides will be discussed in the light of this evidence and final conclusions will be drawn.

THE CASE FOR SOURCE CODE AVAILABILITY

The process of peer review

The claim of open source advocates that this type of software is more secure than proprietary software is based primarily on the perceived strength of the peer review process. According to this process, code is openly published and thus can, and will, be reviewed by other programmers. These programmers, whether specifically looking for bugs or not, will find problems in the code and report them. These bug reports (which will include security bugs) allow the developers to fix the problems and thus the quality of the software improves. Often the person who finds the bug will also submit a 'patch' to solve the problem which serves to speed up the process. The claim is that 'given enough eyeballs, all bugs are shallow' (Raymond, 2000). Some open source advocates claim that this process is in the tradition of the scientific method whereby results are openly published and are not accepted until they have been both reproduced and subjected to the analysis and criticisms of their peers. Despite many differing opinions being put forward regarding the effectiveness of this process, there is certainly evidence that some users perform auditing work to verify for themselves the security of the software they use. In particular, for especially security-sensitive software (for example, cryptographic software such as PGP) there is certainly evidence that some users do seek to verify the source code for themselves (Simpson, 1999). Advocates also point to often-repeated anecdotal evidence of poor reliability of many proprietary, closed source systems such as Microsoft Windows in contrast to the good reputations of systems such as GNU/Linux and FreeBSD, and claim that the peer review process is responsible for this. Of course, this argument must always be taken with a grain of salt since systems such as Sun Microsystems Solaris are also generally regarded as extremely reliable while still being closed source.

A popular example of the strength of the peer review process with respect to security is the supposed difficulty involved for an attacker to insert a 'back door' into an open source program. A 'back door' is malicious code either inserted into, or in some way attached, to a legitimate program or system that allows an attacker to easily and covertly bypass existing security mechanisms. For example, a back door in an operating system might let a person log in as the system administrator without needing a password if they connect from a certain IP address. According to its fans, open source makes it almost impossible for an attacker to hide such code in a program when any number of people can easily gain access to and read its source. One good example of this is that, when the FTP site containing Wietse Venema's 'TCP Wrapper' software was broken into and the attackers modified the source code to contain a back door, the malicious code was discovered and corrected within a day (Garfinkel, 1999). Another telling example was the deliberate inclusion of a back door in Borland/Inprise's Interbase database software in 1992 by the authorized and legitimate programmers working at that company. The purpose of the back door was not actually malicious but, had its presence become known to a malicious party, then this would effectively have granted them complete access to all installations of the database system. Interestingly, the issue was discovered and fixed 9 years after being added, but only shortly after the product was made open source (*CERT Advisory CA-2001-01 Interbase Server Contains Compiled in Back Door Account*, 2001). Therefore, while access to the source code is essentially required in order to add a back door, if the code is proprietary software then it will probably remain vulnerable for a very long time. This has caused many to speculate on the implications of the potential insertion of a back door into Microsoft code during the successful penetration of that company's network during 2000.

Another factor in favour of open source that makes it especially difficult to add such malicious code is the nature of the development process often used in these projects. For example, many projects use the Concurrent Versioning System (CVS) program which controls access and changes to the source code such that these can be managed and tracked. Other projects like the Linux kernel only allow submission of modifications to the kernel in the form of 'patches' to existing code which are carefully examined for bugs by trusted developers with long histories of involvement before being incorporated.

Flexibility and freedom

The very nature of open source software in providing its users with additional rights not given with proprietary software allows them certain options that wouldn't otherwise be available. In some instances these choices potentially have significant security consequences.

For example, an organization using a piece of open source software is free to perform their own in-house security auditing work which would not be possible without access to the source code. Of course, not all organizations have the skills or resources to conduct this sort of work but, often for those with extreme or specific security requirements, such work is virtually mandatory. For example, it has been suggested that some countries throughout the world might have a certain distrust of software developed in another country that represents a potential enemy. Such software could easily contain some form of back door or deliberate security flaw that could be exploited by an enemy if the software were used in a sensitive military or intelligence-oriented environment. In fact, it has recently been reported in the online media that Chinese and German authorities may be limiting the use of U.S.-developed binary-only software in certain situations and possibly promoting Linux as a preferred computing platform for similar reasons (McAllister, 2001), (Lettice, 2001). While there is certainly no evidence to confirm that these fears are justified, for many countries that do not produce all their own software the ability to audit the source code of the programs they use in such situations would certainly be very welcome. This is rarely possible with proprietary software.

This concept can also be extended further. While the ability to audit source code where security requirements are extreme or highly specific is a definite advantage, the ability to modify that software to specifically meet these requirements is perhaps even more significant. With free software, the organization has the option of either contributing development resources to collaborate with the independent free software developers or simply making the modifications themselves in-house. One notable recent example of this is the U.S. Government's National Security Agency (NSA) which produced their own version of the Linux kernel with enhancements to enforce certain advanced security requirements pertaining to their organization (*NSA Security-Enhanced Linux* 2000). They have also released the source code to their modifications and the software is still being improved. This would be much harder to achieve with proprietary software, even for an influential organization like the NSA.

Yet another advantage of the flexibility that free software brings is apparent where a security flaw is discovered. When dependent on binary-only software the user is heavily restricted: their only option is to simply wait, vulnerable, for the vendor to release a patch or update. In some situations a vendor may simply decide that it is not worth releasing an update for this particular problem (Payne, 1999). The users of proprietary software are entirely at the mercy of their chosen vendor. On the other hand, users of free software have a variety of options. Even if their vendor does not release an official patch, very often other users will collaborate to develop their own. And, in the unlikely event that this does not happen, the user still has the option of producing their own fix independently. Without access to the source code, these possibilities simply do not exist.

Other miscellaneous arguments

It is generally held that security bugs in open source software tend to get fixed faster than similar problems in proprietary software, although in recent years commercial vendors have definitely improved their response rate. This is almost certainly due to increased media focus on security issues and the negative publicity that often results today from major security failures. One possible explanation for this is that bug fixes in commercial software often have to go through lengthy regression testing before being released. Another reason is that sometimes a proprietary vendor's release timetable for a new security patch may be influenced more by business issues than technical ones. For example, official acknowledgement of a security problem may result in bad publicity and exert a negative effect on the company's stock price. Some vendors may also be required to privately inform their larger customers about the issue going public and this potentially leaves many ordinary users vulnerable for an unnecessarily extended period of time. In many ways this is a serious disadvantage for closed source software since the longer a known bug is left unfixed, the more likely it is that damage will be done (Schneier, 2000a). Perhaps the classic example of this relates to the infamous 'Ping of Death' bug which affected a large number of operating systems' TCP/IP stacks. Systems all over the Internet were being crashed with oversized ICMP ECHO packets and many users of commercial systems suffered an anxious wait while their vendors implemented and released the appropriate patches. In contrast, fixes were made available for Linux and BSD-based operating systems within hours (Moody, 1997). A similar and related fact is that it is often easier for users who discover security-related bugs to report them directly to the relevant programmers when dealing with open source development projects. In contrast, reporters of bugs in proprietary software often have to deal with multiple layers of corporate bureaucracy and are usually unable to speak directly with the programmers responsible for writing and fixing the code.

Another, perhaps slightly more tenuous, argument in favour of open source is that, while the development of proprietary software is often controlled by commercial imperatives, free software can be developed in accord with purely technical requirements. For example, commercial vendors are often under pressure to rush products to market in order to get an advantage over a competitor. Their inclusion of features is generally dominated by perceived customer requirements. If there is not a strong indication that the software's users will appreciate a given feature then it probably won't be implemented. Unfortunately, security often falls into this category. Generally users want faster, more powerful and easier to use systems. Security typically requires significant investment of resources to be properly implemented and is often simply too intangible to be easily marketable. On the other hand, the vast majority of open source software projects are free to make decisions purely on the basis of their technical merits. If security is required then it can be added – regardless of whether users are demanding it or not. Thus, its advocates argue, free software can be developed according to purely technical mandates which has a positive overall impact on security.

Proponents of closed source software argue that, because the source code for such products is unavailable, it is harder for malicious users to find security bugs in them compared with software where the source code is available (this argument will be explored in more detail in the section 'Vulnerabilities are harder to find'). However, some advocates of open source software have suggested that the truth of this statement is not as straightforward as it seems. Instead, it is argued, bugs in binary-only software can often be found just as easily as in open source counterparts simply by using different techniques. While a common approach to finding some specific classes of security bugs in source code is to perform a search for a set of functions that frequently cause security problems (a tactic that appears to be possible only when the source code is available), a similar process can actually be performed with respect to binary programs. The binary code can be disassembled and system calls to the problematic functions can thus be identified. Similarly, external references to dynamically linked modules also reveal this information. Thus, it is argued, while finding security bugs in closed source programs is certainly different to finding them when the source is available, it is not necessarily always significantly harder. Based upon this argument it would seem that open source programs are, in fact, at a significant advantage, since even a novice open source code auditor that comes across a potential problem in a program's code can easily fix this for themselves. In contrast, fixing similar problems in a binary-only program is much, much harder.

THE CASE AGAINST

Vulnerabilities are harder to find

The case against the security of open source software is derived from what some may perceive as simply an entirely different philosophical perspective. While open source users view the source code auditing and peer review processes as vital to eliminating bugs and thus improving the software, advocates of proprietary software base their hopes for security primarily on the principle that, if these bugs are never discovered, there is no actual security problem and they might as well not exist. This poses the philosophical, almost Taoist, question: if a vulnerability exists in a program but no one knows about it, does it really pose a problem? Bruce Schneier describes this as 'Phase 1' of the 'Window of Exposure' (Schneier, 2000a), (Schneier, 2000b). The closed source school of thought says that since security bugs are easier to discover when you have the source code, restricting access to the source code will reduce the number of security flaws discovered (at least by external parties) and, since flaws that remain undiscovered have no impact on the security of the system, a closed source system will therefore be more secure.

Naturally this hinges on the assumption that vulnerabilities are actually harder to find without having access to the source code. Although this is certainly intuitively correct we will explore the reasoning behind it here for the sake of completeness. It is interesting to note that this is a virtually identical argument to the one made by open source advocates; availability of source code makes it easier to find security bugs. The difference in opinion lies with whether or not this is a positive thing.

Clearly, without access to the source a potential code auditor (whether their intent be good or evil) must make certain guesses about the program's implementation, and even elements of its design, which would otherwise be obvious. Thus searching for security bugs often involves 'black box' style testing where the auditor 'probes' the program with carefully selected input data in order to observe outcomes. For example, they might supply a privileged program running on an Intel platform with an overly long parameter made up of 1,000 ASCII 'A' characters and observe the outcome. If the program crashes and leaves the EIP register filled with the hexadecimal value 0×41 (the number value for the character 'A' in ASCII) then this is a good indication that the program is vulnerable to a 'stack smashing' style buffer overflow attack and the auditor would probably continue to probe the program to confirm exploitability (Levy, 1996). However, this process is clearly very 'hit and miss' and, if the auditor does not get the outcome they were looking for then it is a matter of laboriously back-tracking and trying another approach. On the other hand, with the source code available, the auditor could go straight to the code in question and establish very quickly whether or not it is vulnerable. For the buffer overflow example just given, this dramatically increases the ease of performing the attack since the set of functions frequently responsible for this problem can be searched for quickly (Friedrichs, 2000). Furthermore, the auditor can determine the exact length of the data required to overflow the buffer and other specifics concerning the nature of the injected malicious data that might not otherwise be able to be predicted.

Another possible approach would be to disassemble the binary code. This has the advantage of taking most of the guesswork out of the auditing process since it gives the auditor a more complete view of the program and it's behaviour than even the source code would. However, this does not necessarily make the auditing process significantly easier, since the level of detail provided is often, in fact, far too great. Comprehending large programs that have been converted into assembly language is a formidable task made worse by the optimizations modern compilers often utilize which change the structure of the program logic from how it is in the high-level source code. Thus disassembling frequently does not help in auditing programs for security bugs and may actually involve more effort than simple black-box testing.

Flaws in the 'peer review' argument

So, according to the argument just presented, if hiding the source code inhibits the discovery of new security problems, conversely, publishing the source code must aid the discovery of such problems. Thus, it would appear that the argument for concealing the source actually supports the argument in favour of peer review. However, the differing perspectives held by both camps lie not only in the semiphilosophical difference of whether a system is more secure when vulnerabilities are found and fixed or if they remain hidden, but also in the practical effectiveness of open source code auditing. In essence, is the 'peer review' argument actually valid?

The first question that should be asked is: is publishing the source code of a program effectively the same as publishing new scientific results? For one thing, new scientific results are not accepted or relied upon until they have been comprehensively reviewed, analysed, criticized and replicated. Despite the mass media reports immediately following its 'discovery', no car owners rushed to upgrade their automobiles to a new cold fusion-powered model. In contrast, newly released open source software is very frequently relied upon almost immediately, and before many people have had the chance to conduct even a cursory examination. In fact, often the peer review process does not actually gain much momentum until a significant number of suitably qualified people have taken this initial risk. While some users may have the time and resources to install new software only on special isolated 'testing' networks while they trial and audit it, many users simply have to trust the code (whether open source or otherwise) and install it on their production systems. As such, the situation for these users is much the same as if they were running proprietary software and, in fact, commercial vendors would certainly argue that they themselves are uniquely positioned to be able to invest resources into such testing and auditing regimes in a way that most users cannot. This begs the question: if the source is available to all but very few are actually looking, how big is the peer review advantage really? (Levy, 2000). Thus suggestion is that users of open source software all believe that everyone else is auditing the source code for them and so do not bother to do so themselves (Viega, 2000). In particular, with the many various Linux distributions becoming more and more popular, most software, even for open source systems, is increasingly distributed primarily in binary form. Although the source code is still available, this usually has to be downloaded separately and, with changing user demographics, fewer and fewer users are bothering to do this. While once the majority of Linux users preferred to obtain the source code themselves and compile and install it by hand, toady most simply want to use the software without going to this additional trouble. Furthermore, this new group of users also tend to have much less background, let alone interest, in security.

Consequently, evidence abounds that the peer review process is not always as effective as is often argued. Despite Raymond's claim that 'given enough eyeballs, all bugs are shallow', security bugs have often proved to be extremely 'deep' (Raymond, 2000). For example, (Viega, 2000) cites bugs in the GNU Mailman program that existed for 3 years without being discovered. Furthermore, both that paper and (Garfinkel, 1999) discuss buffer overflows in MIT's Kerberos distribution that lay undiscovered for over 10 years (Neuman and Ts'o, 1994). Interestingly (Garfinkel, 1999) also considers the example of the SENDMAIL SMTP server which for years was regularly plagued with security problems. Here Garfinkel notes that SENDMAIL's security record improved dramatically after a company was formed to develop and market a commercial version of the software. To advocates of proprietary software these examples clearly demonstrate the security failures of open source and the peer review process. They claim that users of open source software are, quite literally, being lulled into a false sense of security.

So assuming that there are ‘enough eyeballs’, why are these bugs not always found? One possible answer is that even though many people are examining the source, not all are necessarily qualified to properly identify security problems they might find. In the realm of scientific peer review, the reviewers themselves tend to have similar levels of experience and expertise, however, this is frequently not the case with respect to software. Although most security issues are not difficult for skilled programmers to understand in concept, in practice actually auditing source code for them may not be so easy. Having experience is important, however, there are few (if any) comprehensive documents available to teach potential code auditors how to do this work effectively. As a result, gaining such experience may be problematic. Experienced code auditors almost seem to advocate a cavalier ‘jump in the deep end and see if you sink or swim’ attitude. To many potential novice code auditors this may seem particularly discouraging.

A related issue is that effective auditing often requires not only an understanding of the programming language concerned, but also quite possible a strong background in other specific areas. This may be a challenge in itself if the language is not a widely used one. For example the GNU Mailman bug cited previously occurred in Python code and it is believed that the lower popularity of this language as compared with, for example, C contributed to it remaining undiscovered for so long (Viega, 2000). For example, to effectively audit network servers the auditor must have a good understanding of the relevant networking protocol. Without this it may be hard for an auditor to gain a proper understanding of the code being studied. Understanding the underlying protocol in detail would probably be necessary to properly analyse the flow of data throughout the program in order to determine potential vectors for an attacker to inject specially crafted malicious data into the running server and exploit a particular vulnerability. If an auditor does not understand the protocol then they will have trouble telling whether the source data in a suspicious strcpy () call is actually derived from an untrusted source or not. This problem is even more extreme when dealing with specialized areas such as cryptography. The auditor must not only understand the programming language and the protocol in use, but they must also have a very solid understanding of the wide range of issues related to securely implementing cryptographic software. A recent example of this would be a vulnerability found in some implementations of the Secure Shell (SSH) remote login system protocol version 1.5 (Ylönen 1996), (*SSH1 Session Key Retrieval Vulnerability*, 2001). Finding this vulnerability required not only an understanding of the SSH protocol itself, but also of many advanced issues relating to cryptanalysis and secure implementation of cryptographic software. Very few ‘eyeballs’ could be expected to have such qualifications.

However, even assuming that there are sufficient numbers of auditors with the relevant skills, having the source code still does not represent a complete documentation of the program. The source code represents one possible implementation of a particular design, however, it does not necessarily make it easy to comprehend and properly analyse all aspects of that design. While many security problems are caused by what amounts to coding errors, there are also often flaws in the program's design. The source code by itself does not explain the rationale behind certain design decisions so, unless the auditor has access to complete and detailed documentation concerning the program's design then errors pertaining to this may be hard to find.

A final argument made against the validity of the open source peer review process is that, while open source code is certainly being read, in many cases the only people with the requisite security background needed to find holes are those whose motives are not actually entirely honest (Levy, 2000). It is argued that these so-called 'black hats' or 'backers' are willing to search source code that is easily available in order to find security bugs that they can exploit while, at the same time, being less interested in closed source programs where much more effort is required. Thus this represents the converse of the argument given in the section 'Vulnerabilities are harder to find'. The true validity of this argument is somewhat hard to determine but it does seem to apply at least some of the time given that some vulnerabilities actually become public by virtue of 'attack tools' found on compromized machines and their description in 'underground' publications.

Other miscellaneous arguments

One interesting argument made in (Levy, 2000) is that it can never be guaranteed that the source code being read actually corresponds to the binary version of the program that is then used. The example is given from Thompson (1984) where a compiler is created that detects the source code for a certain program (in this case, the Unix login program) and covertly inserts a back-door into the generated object code. Furthermore, the compiler also detects when it is compiling another new version of itself and automatically inserts the code to perpetuate the back-door. This creates the situation where a back-door perpetually exists but there is no evidence of it in the program's source code. Such a problem would be very hard to find and theoretically could exist almost indefinitely. This is a rather sophisticated attack and there is no evidence that such a trick has ever actually been perpetrated. Indeed, open source advocates would very likely argue that the idea is somewhat far-fetched and, while possible, would probably depend on the dishonesty of a key member of the development staff from the particular vendor. Furthermore, if such an accusation could be levelled at open source software, certainly the danger is at least as great with binary-only software. If an attacker were able to gain access to proprietary code and add such a back-door then it would be even less likely to be discovered (as discussed in the section 'The process of peer review'). However, given that open source software is distributed predominantly in binary form, such an attack is still entirely possible. Users of Linux distributions, for example, have little guarantee that the binary packages they implicitly trust every day bear any relationship to the source code versions their distributor makes available for download. Since the source code for these programs is available, an attacker could easily create a modified version without resorting to the laborious tasks of disassembly and binary patching which would be necessary if the same attack was attempted against most proprietary software.

EMPIRICAL RESULTS

Although much has been written concerning the variety of opinions on the security of open source software, very little empirical or factual data has been presented. In his paper, Viega perhaps comes the closest by presenting and analysing his own experiences with security bugs that he unwittingly introduced into the GNU Mailman software (Viega, 2000). However, even this represents a purely qualitative discussion of a single piece of software.

However, quantitative results do exist that shed some light on the various claims made by both camps regarding this issue. A study from 1999 took three similar Unix-based operating systems, studied the consideration of security during their development, and compared this against a quantitative assessment of their respective security (Payne, 1999). The three systems considered were Sun Microsystems Solaris, Debian GNU/Linux and OpenBSD – of these, the latter two are both open source systems. The purpose of this work was purely and simply to analyse the role that the development process plays in the effective security of fielded systems (Payne, 2000), however, since whether a piece of software is open source or not is a significant aspect of the program's development, these results can be used when assessing the relationship between open source code and security.

While the details of the metric used to assess security are too involved to be reviewed comprehensively here, the technique involved an examination of each system's security-related features and known vulnerabilities. (Payne, 1999) contains a complete description of the metric, as well as a comprehensive discussion of its development, application, results (including details of all the security features and vulnerabilities studied) and validity, while (Payne, 2000) contains a brief overview. Each of these was assigned a quantitative score representing either the positive effect (in the case of security features) or severity of threat (for the vulnerabilities). Features were rated based upon their effectiveness and importance while vulnerabilities were scored according to the extent of their impact, the ease of exploiting the flaw and the difficulty of solving the problem. Each item was then classified into one or more of four security 'dimensions': confidentiality, integrity, availability and audit. These four were based upon the three key properties of security (confidentiality, integrity and availability) plus the common additional requirement that a system keep a record of security-related events through some logging mechanism (Pfleeger, 1997), (Garfinkel & Spafford, 1996), (U.S. Department of Defence (DOD), 1985), (Russell & Gangemi Sr., 1992). Confidentiality features studied included mechanisms such as cryptographic file systems or other data encryption systems. Some of the operating systems considered included the ability to prevent processes from modifying files in certain ways, even if the process were executing with superuser privileges and this acted as an integrity-based security feature. The systems also included security controls specifically relating to the availability dimension which allowed the amount of resources utilized by each process to be specified and limited. The objective here is to prevent a malicious user from over-utilizing a shared resource and denying other users access. Different logging mechanisms were often included such as special software to log network packets received by the computer or to log information about the system call requests made by processes to the kernel. These are examples of some of the availability features included in the systems studied.

Table 1. Summarized results from the security analysis

	Debian	Solaris	OpenBSD
Features			
Confidentiality	5.90	6.08	7.50
Integrity	5.88	6.17	7.38
Availability	7.00	5.75	6.00
Audit	6.90	5.67	7.25
Number	15	11	18
Average	6.42	5.92	7.03
Vulnerabilities			
Confidentiality	6.75	8.13	4.50
Integrity	7.70	7.40	4.25
Availability	8.10	7.00	8.00
Audit	8.33	8.42	0.00
Number	12	21	5
Average	7.72	7.74	4.19
Unscaled Score	–1.30	–1.80	2.80
Scaling Factor	1.25	0.52	3.60
Final Score	–1.0	–3.5	10.2

For each item considered, the analyst was required to extensively document both information about the feature or vulnerability and to justify the scores assigned to it, the classification it received and also its inclusion in the analysis. Once all items had been identified, documented and rated, scores for both the positives and negatives for each dimension were then averaged (Payne, 1999).

The overall score for each system was then calculated by combining the results for each system's features and vulnerabilities and scaling this overall score in accord with the relative number of these. The scaling factor used represents the ratio of features to vulnerabilities. A summarized overview of those results is presented in Table 1. Note that the results from this metric are essentially multi-layered in that they can be viewed at varying levels of detail. The results provide an overall, single-value result for each system allowing for a rather broad comparison between systems but they can also provide significantly more information when considered more closely. By examining the various figures that were used to produce these overall results, a rather rich and detailed picture can be produced of the security strengths and weaknesses of the various systems. Indeed, this property of the metric used is what makes the results from this previous study useful in answering the question of whether or not open source software has any security advantage, despite the fact that the original study was not specifically developed for this purpose.

The results show that, of the three systems, OpenBSD had the most number of security features (18) with Debian second (15) and Solaris third (11). Of these features, OpenBSD's features rated highest scoring 7.03 out of 10 while Debian's scored 6.42 and Solaris' scored 5.92. A similar pattern was observed for the vulnerabilities with OpenBSD having the fewest (5). These vulnerabilities were also relatively minor only rating an average of 4.19 out of 10. Note that the score of '0.00' for OpenBSD's audit vulnerabilities is because this system had no actual availability vulnerabilities. In general, availability vulnerabilities were comparatively rare and so it is not surprising that OpenBSD had none given that it only had five recorded problems in total. Debian had the next fewest with 12 vulnerabilities rating an average of 7.72 while Solaris had the most with 21 vulnerabilities averaging 7.74 out of 10. The final scaled results gave OpenBSD the best score of 10.2 while Debian scored –10 and Solaris –3.5. The research concluded that the primary reason for OpenBSD's success was the consistent focus on security throughout all phases of the development process and, in particular, the project's pro-active code auditing work which resulted in a large number of potential problems being eliminated long before they were discovered and reported amongst the wider community (Payne, 2000). We will now discuss these results in more detail with respect to the question of whether open source software is more secure than proprietary code.

DISCUSSION

Before analysing the relative merits of the cases presented by both sides in the light of the empirical evidence above, it is useful to briefly review the respective arguments. On the one hand, those who argue in favour of open source software providing superior security, point to the peer review process that supposedly occurs when the source code for a program is openly published. Undoubtedly this assists in the discovery of bugs, some of which certainly will be security bugs. The source code availability and, in particular, the freedoms granted through 'open source' licences allow security-conscious users to not only perform their own code audits, but also to add any additional security features that they require. In this way open source software creates the opportunity for provision of enhanced security functionality and assurance for organizations with a policy that demands such security. Finally, open source software gives users additional flexibility in their response to the discovery of a new security problem. While users of proprietary software are entirely dependent on their vendor for releasing a fix for the problem, open source users can even patch their own software if required.

On the other hand, those in favour of proprietary, binary-only software present an entirely different view. They argue that, if vulnerabilities are not discovered then they effectively don't exist, thus making the software secure. Since it will typically be harder to find security bugs in binary-only programs this makes software of this type more secure than its open source counterparts. Additionally, those advocating proprietary software argue that the peer review process, which open source software relies on for its security, is not actually as effective as some claim. Not as many people audit the code for security flaws as is often thought and these people frequently don't have the expertise required to identify some of the more subtle problems. In fact, they argue, of all those who search the source code for new security problems, many with the expertise to identify such problems are actually potential system attackers who probably will not report the problem to the wider community to allow it to be fixed. In this way, publishing the source code makes the system more vulnerable to those who seek to attack it while not making it significantly more secure.

What is interesting is that these latter two arguments appear to come very close to contradicting one another. The first argument says that it is harder to find bugs in binary-only code and, consequently, easier to find them in open source code. The second argument says that the peer review process is not particularly effective and many bugs won't actually be found. To some, this is hard to reconcile. After all, if it is easy to find bugs in open source software then surely the peer review process must actually work? Conversely, if the peer review process is limited in effectiveness then perhaps open source software can also lay claim to some of the assumed advantages of binary-only software in terms of the difficulty of finding bugs? The answer to resolving this contradiction lies in acknowledging that, while it is certainly easier to find simple or obvious security bugs in open source code, it is often much harder to find the more subtle ones. Naturally they are even less likely to be found in closed source software. The outcome of the debate, therefore, appears to rest on the question of whether or not the code is actually reviewed by users who not only have at least a basic security background, but are also willing to report the flaws they find in order to have them fixed. If this auditing work is not actually conducted by such 'honest' users then it will almost certainly be done by those with malicious motivations who will use any vulnerabilities they find for their own purposes.

Another interesting fact concerning both the primary arguments for and against open source security is that they argue essentially the same thing. The peer review argument suggests flaws will be found (and fixed) much more quickly while the binary-only code argument claims that flaws will be much harder to find in order to be exploited by malicious parties. Thus, they both argue that source code availability makes it easier to find security problems (Payne, 1999). The open source view point is that finding and fixing bugs leads to an overall decline in the number of bugs in the code over time, while the proprietary view point is that it is better to simply try and hide potential bugs which thus prevents them from being exploited. The origin of this largely philosophical difference in opinion probably lies with the predominantly commercial nature of proprietary software. Whereas open source software can endure the occasional rash of security problems being made public, commercial software usually depends on maintaining a positive public image to ensure continuing sales and a healthy stock price. The discovery of new security problems can have a negative effect on both of these and so it makes more sense to try and prevent these problems from being discovered. Therefore the question for commercial software vendors is, does publishing the source code provide significant enough security benefits to warrant this regardless of the potential for short term public-relations pain?

We will now seek to answer these two questions based upon the results of the study previously cited. Note that although the study was empirically based and its results certainly suggest a conclusion regarding the topic at hand, these conclusions are far from being final. The previous research studied only three systems and over the span of a single version release. To further confirm these results a more detailed and lengthy study would need to be conducted covering more systems over a longer period of time. Additionally, more work would be required to further develop and refine the metric used to assess the security of the three systems.

Firstly a comparison of the final score for Debian and Solaris shows the latter (a proprietary system) scored –3.5, placing it behind Debian (an open source system) on –1.0, although clearly the relative difference is far from large (see Table 1). Therefore, the obvious initial answer is that open source software is probably more secure than proprietary software although the advantage is not as great as some might claim. However, taking advantage of the multi-layered results from the metric and examining these more closely yields something even more interesting. While binary-only software is supposed to limit the ability for vulnerabilities to be found, the version of Solaris studied recorded 21 vulnerabilities while Debian had only 12! The results therefore suggests that hiding the source code does not necessarily bring about the degree of protection that might be anticipated. While a program may effectively be secure if its vulnerabilities remain in Phase 1 of Schneier's Window of Exposure, once the flaw is discovered and Phase 2 begins, the situation becomes much worse (Schneier, 2000a). Exactly how bad depends entirely on who it is that has discovered the vulnerability. If the problem has been discovered by a system 'cracker' then the situation is bleak indeed. Perhaps the difference between open source and proprietary code at this point is that the vulnerability is more likely to have been discovered by an individual who will fix the problem or report it to the vendor if the software is open source. It is less likely that the vulnerability will be found by an 'honest' user who merely stumbles across it if the code is not available. On the other hand, malicious parties are much more inclined to closely examine and probe a piece of software specifically in order to discover a vulnerability and, as the empirical results cited suggest, if there are large numbers of serious security problems present in a program then it is very likely these will be discovered if looked for.

Based on these results it would appear that open source systems tend to be more secure, however, the results from the security analysis of the third system, OpenBSD, have not yet been considered. If the security of a system were heavily dependent on its status as either an open source or proprietary system then the result for either Debian or OpenBSD would clearly be anomalous. In scoring 10.2, OpenBSD was the only system of the tree to receive a positive score and, a comparison with the magnitudes of the other two scores suggests this is a relatively high score also. Therefore, the significant differences between Debian and OpenBSD's score support the argument that making a program 'open source' does not, by itself, automatically improve the security of the program (Levy, 2000), (Viega, 2000). What, therefore, accounts for the dramatically better security exhibited by the OpenBSD system over the other two? The author believes that the answer to this question lies in the fact that, while the source code for the Debian system is available for anyone who cares to examine it, the OpenBSD source code is regularly and purposefully examined with the explicit intention of finding and fixing security holes (Payne, 1999), (Payne, 2000). Thus it is this auditing work, rather than simply the general availability of source code, that is responsible for OpenBSD's low number of security problems. This point bears repeating: software will not become automatically more secure by virtue of its source code being published. It requires a careful, focused code audit by programmers with the necessary background and security expertise to make a significant impact on the overall security of a body of code. Thus, there is not reason why this work could not be done internally within a proprietary software company given sufficient allocation of resources and personnel. In fact, in interviews conducted as part of this previous research, a leading OpenBSD developer claimed that the availability of source code had had little effect on the security of the system. Instead, a relatively small number of key developers had been chiefly responsible for the removal of security-related bugs from the code base (Payne, 1999). If this is true (and it does appear to be supported by the empirical evidence), then clearly the opportunity exists for proprietary developers to dramatically improve the security of their software without needing to release the source code.

Open source software often lays claim to being more stable than proprietary code since it allows the code to be actively reviewed by its users. However, where security is concerned, clearly this is not necessarily the case. Security flaws are not like ordinary bugs. They are subtle and complex, and often require specialized or in-depth knowledge to identify. Source code auditing for the purpose of security is not the same as simply reading or experimenting with the code in order to find ordinary bugs. It requires special expertise and a particular focus. Thus, regardless of other security-related advantages such as the ability to adapt to specialized security requirements and freedom from vendor dependence as described in the section ‘Flexibility and freedom’, open source software is not intrinsically more secure than proprietary code. To quote Elias Levy, ‘Open source software certainly does have the potential to be more secure than its closed source counterpart. But make no mistake, simply being open source is no guarantee of security’ (Levy, 2000).

REFERENCES

CERT Advisory CA-2001-01 Interbase Server Contains Compiled in Back Door Account (2001). http://www.cert.org/advisories/CA-2001-01.html.

Chowdhry, P. (1999) Open source meets the ‘Baywatch’ factor. http://www.zdnet.com/eweek/stories/general/0,11011,2352305,00.html.

DiBona, C., Ockman, S. & Stone, M. (eds) (1999) *Open Sources: Voices from the Open Source Revolution*. O’Reilly & Associates, Sebastapol, California.

Friedrichs, O. (2000) Secure programming. http://www.securityfocus.com/forums/secprog/secure-programming.html. Version 1.00.

Garfinkel, S. (1999) Open source: how secure? http:/www.wideopen.com/story/101.html.

Garfinkel, S. & Spafford, E. (1996) *Practical Unix and Internet Security*, 2nd edn. O’Reilly & Associates, Sebastapol, California.

Gross, G. (2000) Panel: open source security needs to be a priority. http://www.newsforge.com/article.pl?sid=00/10/17/1830254.

Lettice, J. (2001) German armed forces ban MS software, citing NSA snooping http://www.theregister.co.uk/content/4/17679.html.

Levy, E. (1996) Smashing the stack for fun and profit. *Phrack 49*.

Levy, E. (2000) Wide open source. http://www.securityfocus.com/commentary/19.

McAllister, N. (2001) The spy who hacked me: will open source be the hero of international security. http://www.sfgate.com/cgibin/article.cgi?file=/technology/archive/2001/03/15/china.dtl.

Moody, G. (1997) The greatest OS that (n)ever was. http://www.wired.com/wired/5.08/linux_pr.html.

Netcraft Web Server Survey (2001) http://www.netcraft.com/survey/.

Neuman, B. C. & Ts’o, T. (1994) Kerberos: An authentication service for computer networks. *IEEE Communications* **32**(9): 33–38.

Norin, L. & Stöckel, F. (1998) Open-source software development methodology. http://www.ludd.luth.se/users/no/mssc_abstract.html.

NSA Security-Enhanced Linux (2000) http://www.nsa.gov/selinux/.

Payne, C. (1999) *Security through design as a paradigm for systems development*. Murdoch University, Perth, Western Australia.

Payne, C. (2000) The role of the development process in operating system security. In: *Information Security: Third International Workshop, ISW 2000, Vol. 1975 of Lecture Notes in Computer Science*. Pieprzyk, J., Okamoto, E. & Seberry, J. (eds), pp. 277–291 Springer, Germany.

Pfleeger, C. (1997) *Security in Computing*. Prentice-Hall, Upper Saddle River, New Jersey.

Raymond, E. (2000) The Cathedral and the Bazaar. http://www.tuxedo.org/esr/writings/cathedral-bazaar/.

Russell, D. & Gangemi Sr., G. (1992) *Computer Security Basics*. O'Reilly & Associates, USA.

Schneier, B. (2000a) Closing the window of exposure: reflections on the future of security. http://www.securityfocus.com/templates/-forum_message.html?forum=2&head=3384&id=3384.

Schneier, B. (2000b) Full disclosure and the window of exposure. *Crypto-Gram*. http://www.counterpare.com/crypto-gram-0009.html#1.

Simpson, S. (1999) PGP DH vs RSA FAQ. http://www.scramdisk.clara.net/pgpfaq.html.

SSH1 Session Key Retrieval Vulnerability (2001) http://www.securityfocus.com/vdb/bottom.html?vid=2344.

Thompson, K. (1984) Reflections on trusting trust. *Communications of the ACM,* **27**.

U.S. Department of Defence (DOD) (1985) *Trusted computer system evaluation criteria*. DOD 5200.28-STD.

Viega, J. (2000) The myth of open source security. http:developer.earthweb.com/journal/techfocus/052600_security.html.

Ylönen, T. (1996) SSH – secure login connections over the Internet. *Proceedings of the 6th USENIX UNIX Security Symposium*.

Biography

Christian Payne began his undergraduate university career as a scholarship student in Chemistry who also happened to be majoring in Computer Science. Five years later he emerged with a first class honours degree in Computer Science and a deep fascination for research into ways and means of securing computer systems. He is now one and a half years into completing his PhD exploring his specific interests of applied cryptography and operating system security. He has also recently written, and currently teaches, a new undergraduate course on computer security at Murdoch University, Western Australia.